Symbolic Range Analysis of Pointers
Dissertation Summary

Vitor Mendes paisante
Advisor: Fernando Magno Quinto Pereira
Co-advisor: Leonardo Barbosa e Oliveira

[1]Federal University od Minas Gerais Computer Science Department

{paisante}@dcc.ufmg.br

Abstract. *Alias analysis is one of the most fundamental techniques that compilers use to optimize languages with pointers. However, in spite of all the attention that this topic has received, the current state-of-the-art approaches inside compilers still face challenges regarding precision and speed. In particular, pointer arithmetic, a key feature in C and C++, is yet to be handled satisfactorily. This work presented a new alias analysis algorithm to solve this problem. The key insight of our approach was to combine alias analysis with symbolic range analysis. This combination allowed us to disambiguate fields within arrays and structs, effectively achieving more precision than traditional algorithms. To validate our technique, we implemented it on top of the LLVM compiler. Tests on a vast suite of benchmarks showed that we could disambiguate several kinds of C idioms that current state-of-the-art analyses could not deal with. In particular, we could disambiguate 1.35x more queries than the alias analysis that was currently available in LLVM. Furthermore, our analysis was very fast: we were able go over one million assembly instructions in 10 seconds..*

1. Introduction

Pointer analysis is one of the most fundamental compiler technologies. This analysis lets the compiler distinguish one memory location from others; hence, it provides the necessary information to transform code that manipulates memory. Given this importance, it comes as no surprise that pointer analysis has been one of the most researched topics within the field of compiler construction[Hind 2001]. This research has contributed to make the present algorithms more precise [Hardekopf and Lin 2007, Zhang et al. 2014], and faster [Hardekopf and Lin 2011, Shang et al. 2012]. Nevertheless, one particular feature of imperative programming languages remained to be handled satisfactorily by the current state-of-the-art approaches: the disambiguation of pointer intervals.

Mainstream compilers still struggle to distinguish intervals within the same array. In other words, state-of-the-art pointer analyses often fail to disambiguate regions addressed from a common base pointer via different offsets, as explained by Yong and Horwitz [Yong and Horwitz 2004]. Figure 1 shows an example that state-of-the-art pointer analyses tend not to deal with satisfactorily. In the example, both stores on the r array posses different offsets and they do not alias. To figure out that such offsets are disjoint it is necessary to verify their ranges and add a new layer of analysis to the current pointer analyses available.

```
char* duplicate (int size, char* v) {
  if(size > 0) {
    char* r = malloc(size*2);
    int i;
    for(i = 0; i < size; i++) {
      r[i] = v[i];
      r[i+size] = v[i];
    }
  }
  else return NULL;
}
```

Figura 1. Example that state-of-the-art pointer analyses handle unsatisfactorily.

Field-sensitive pointer analysis, provide a partial solution to this problem. These analyses can distinguish different fields within a record, such as a struct in C [Pearce et al. 2004], or a class in Java [Yan et al. 2011]. However, they rely on syntax that is usually absent in the low level program representations adopted by compilers. Shape analyses [Jones and Muchnick 1982, Sagiv et al. 1998] can disambiguate subparts of data-structures such as arrays, yet their scalability remains an issue to be solved. Consequently, many compiler optimizations, such as loop transformations, tiling, fission, skewing and interchanging [Wolfe 1996, Ch.09], are very limited in practice. Therefore, we claim that, to reach their full potential, compilers need to be provided with more effective alias analyses.

This dissertation described such an analysis. We introduced an abstract domain that associates pointers with symbolic ranges. In other words, for each pointer p we conservatively estimate the range of memory slots that can be addressed as an offset of p. We let $> (p)$ be the global abstract address set associated with pointer p, such that if $\mathsf{loc}_i + [l, u] \in > (p)$, then p may dereference any address from $@(\mathsf{loc}_i) + l$ to $@(\mathsf{loc}_i) + u$, where loc_i is a program site that contains a memory allocation call, and $@(\mathsf{loc}_i)$ is the actual return address of the *malloc* at runtime. We let $\{l, u\}$ be two *symbols* defined within the program code. Like the vast majority of pointer analyses available in the compiler literature, from Andersen's work [Andersen 1994] to the more recent technique of Zhang *et al.* [Zhang et al. 2014], our method is correct if the underlying program is also correct. In other words, our results are sound with respect to the semantics of the program if this program has no undefined behavior, such as out-of-bounds accesses.

The key insight of our research was the combination of pointer analysis with range analysis on the symbolic interval lattice. In a symbolic range analysis, ranges are defined as expressions of the program's symbols, a symbol being either a constant or the name of a variable. There exist many approaches to symbolic range analyses in the literature [Blume and Eigenmann 1994, Nazaré et al. 2014, Rugina and Rinard 2005]. The algorithms that we presented in this work do not depend on any particular implementation. Nevertheless, the more precise the range analysis that we use, the more precise the analysis facts that we produce. In this work we adopted the symbolic range analysis proposed in 1994 by William Blume and Rudolf Eigenmann [Blume and Eigenmann 1994].

```
1  #include <stdlib.h>
2
3  void prepare(char* p, int N, char* m) {
4    char *i, *e, *f;
5    for (i = p, e = p + N; i < e; i += 2) {
6      *i = 0;
7      *(i + 1) = 0xFF;
8    }
9    for (f = e + strlen(m); i < f; i++) {
10     *i = *m;
11     m++;
12   }
13 }
14
15 int main(int argc, char** argv) {
16   int Z = atoi(argv[1]);
17   char* b = (char*)malloc(Z);
18   char* s = (char*)malloc(strlen(argv[2]));
19   strcpy(s, argv[2]);
20   prepare(b, Z, s);
21   ...
22   return 0;
23 }
```

Figura 2. Example of program that builds up messages as sequences of serialized bytes. We are interested in disambiguating the locations accessed at lines 6 and 10.

2. Analysis Overview

We have two different ways to answer the following question: "do pointers tmp_i and tmp_j alias?"These tests are called *global* and *local*. In this section, we will use two different examples to illustrate situations in which each query is more effective. These distinct strategies are complementary: one is not a superset of the other.

Global pointer disambiguation. Figure 2 illustrates our first approach to disambiguate pointers. The figure shows a pattern typically found in distributed systems implemented in C. Messages are represented as arrays of bytes. In this particular example, messages have two parts: an identifier, which is stored in the beginning of the array, and a payload, which is stored right after. The loops in lines 5-8 and 9-12 fill up each of these parts with data. If a compiler can prove that the stores at lines 6 and 10 are always independent, then it can perform optimizations that would not be possible otherwise. For instance, it can parallelize the loops, or switch them, or merge them into a single body.

No alias analysis currently available in either gcc or LLVM is able to disambiguate the stores at lines 6 and 10. These analyses are limited because they do not contain *range information*. The range interval $[l, u]$ associated with a variable i is an estimate of the lowest (l) and highest (u) values that i can assume throughout the execution of the program. In this work, we proposed an alias analysis that solves this problem. To achieve this goal, we coupled this alias analysis with range analysis on symbolic intervals [Blume and Eigenmann 1994]. Thus, we can say that the store at line 6 might modify any address from $\mathtt{p}+0$ to $\mathtt{p}+\mathtt{N}-1$, and that the store at line 10 might write on any address from $\mathtt{p}+N$ to $\mathtt{p}+N+\mathtt{strlen(m)}-1$. For this purpose, we use an *abstract address* that encodes the actual value(s) of p inside the `prepare` function. These memory addresses

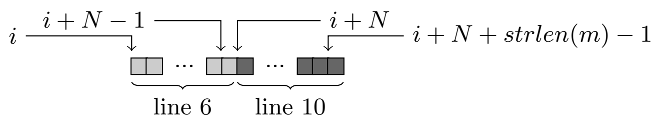

Figura 3. Array p in the routine `prepare` seen in Fig.2. Lines 6 and 10 represent the different stores in the figure.

```
void accelerate
(float* p, float X, float Y, int N) {
  int i = 0;
  while (i < N) {
    p[i] += X;     // float* tmp0 = p + i; *tmp0 = ...;
    p[i + 1] += Y;      // float* tmp1 = p + i + 1;
    i += 2;             // *tmp1 = ...;
  }
}
```

Figura 4. Program that shows the need to assign common names to addresses that spring from the same base pointer.

```
void accelerate
(float* p, float X, float Y, int N) {
  int i = 0;
  while (i < N) {
    float* newp = p+i;    // LR(newp) = locnew + [0, 0]
    newp[0] += X;  // float* tmp2 = newp; *tmp2 = ...;
    newp[1] += Y;  // float* tmp3 = newp + 1; *tmp3 = ...;
    i += 2;
  }
}
```

Figura 5. Program from Figure 4, after pointer is renamed within loop.

are depicted in Figure 3, where each square represents a memory slot.

Whole program analysis reveals that there are two candidate locations that any pointer in the program may refer to. These locations have been created at lines 17 and 18 of Figure 2, and we represent them abstractly as loc_{17} and loc_{18}. These names are unique across the entire program. After running our analysis, we find out that the abstract state (GR) of `i` at line 6 is $\text{GR}(\texttt{i}_{\ell n.6}) = \{\mathsf{loc}_{17} + [0, \texttt{N} - 1]$, and that the abstract state of `i` at line 10 is $\text{GR}(\texttt{i}_{\ell n.10}) = \{\mathsf{loc}_{17} + [\texttt{N}, \texttt{N} + \texttt{strlen}(m) - 1]\}$. Given that these two abstract ranges do not intersect, we know that the two stores update always different locations. We call this check the *global disambiguation criterion*.

Local pointer disambiguation. Figure 4 shows a program in which the simple intersection of ranges would not let us disambiguate pointers tmp_0 and tmp_1. After solving global range analysis for that program, we have that $\text{GR}(tmp_0) = \{\mathsf{loc}_0 + [0, N+1]\}$ and that $\text{GR}(tmp_1) = \{\mathsf{loc}_0 + [1, N+2]\}$, where loc_0 defines the abstract address of the function parameter p. The intersection of these ranges is non-empty for $N \geq 1$. Thus, the global check that we have used to disambiguate locations in Figure 2 does not work in Figure 4. Notwithstanding this fact, we know that tmp_0 and tmp_1 will never point to a common location. In fact, these pointers constitute different offsets from the same base address. To deal with this imprecision of the global check, we will be also discussing a *local disambiguation criterion*. In this case, we rename every pointer p that is alive at the beginning of a single entry region to a fresh name new_p. Whereas we use the global test for pointers in different regions, the local test is applied onto pointers within the same single entry region. After renaming, we update the table of pointer pairs, so that $\text{LR}(new_p) = \mathsf{loc}_{new} + [0, 0]$, regardless of the old ranges assigned to the original pointer p. In Figure 5 we would have that $\text{LR}(tmp_2) = \mathsf{loc}_{new} + [0, 0]$ and $\text{LR}(tmp_3) = \mathsf{loc}_{new} + [1, 1]$, where tmp_2 is the name of the address $new_p[0]$, and tmp_3 is the name of the address $new_p[1]$. This new binding of intervals to pointers gives us empty intersections between similar locations in $\text{LR}(tmp_2)$ and $\text{LR}(tmp_3)$. Consequently, the local check is able to distinguish addresses referenced by tmp_2 and tmp_3.

3. Summary of experimental results

To validate our ideas, we implemented them in the LLVM compilation infrastructure [Lattner and Adve 2004]. We tested our pointer analysis onto three different benchmarks used in previous work related to pointer disambiguation: Prolangs [Ryder et al. 2001], PtrDist [Zhao et al. 2005] and MallocBench [Grunwald et al. 1993]. Our analysis fared linearly on the size of programs. It can go over one-million assembly instructions in approximately 10 seconds. Furthermore, we can disambiguate 1.35x more queries than the alias analyses currently available in LLVM.

4. Summary of publications

The analysis described here was published on the International Symposium on Code Generation and Optimization (CGO) of 2016 held in Barcelona, Spain [Paisante et al. 2016]. Further development of our analysis techniques yielded another published article on the International Symposium on Code Generation and Optimization (CGO) of 2017 held in Austin, Texas [Maleej et al. 2017].

The technology behind it is also present on two other publications [Paisante et al. 2014, Saggioro et al. 2015]. In these papers, the algorithm presented here was used to infer the layout and the content of buffers transfered through the network. This was useful for verifying, in a safe communication line, if the information transfered between two programs through a network should be considered potentially dangerous or not. If proven not dangerous, guards for checking integer overflows may not be necessary. These articles proposed methods for such verification to be used on the internet of things (IoT), where simple devices could run significantly faster with a reduced number of integer overflow checks. The layout and content inference analysis used in these papers differed from our current approach by using a numerical range analysis, since a symbolic approach would not be of relevance for such application.

5. Final Conclusions

In this work we presented a new alias analysis technique that handles, within the same theoretical framework, the subtleties of pointer arithmetic and memory indexation. Our technique can disambiguate regions within arrays and C-like structs using the same abstract interpreter. We have achieved precision in our algorithm by combining alias analysis with classic range analysis on the symbolic domain. Our analysis is fast, and handles cases that the implementations of pointer analyses currently available in LLVM cannot deal with.

Apart from this contribution, there is plenty to study on the area of pointer analysis, and the area of pointer arithmetics still needs quite a bit of research. Their dire needs are very efficient static analyses that run fast on very big programs, and very lean dynamic analyses. Our focus has been on the static analysis side. Lazy implementations, where main computations are made on the query moment, seem to be a very promising take on alias analyses and can expedite runtime. Focus on integrating new proposals to bigger compilation frameworks and existing optimizations should also be explored by the community on an effort of making new technology more usable across researchers and projects. There is still a lot of work to be done and we hope that our contribution can be only a building block of a much bigger effort from many more scientists.

Referências

Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.

Blume, W. and Eigenmann, R. (1994). Symbolic range propagation. In *In IPPS*, pages 357–363.

Grunwald, D., Zorn, B., and Henderson, R. (1993). Improving the cache locality of memory allocation. In *In PLDI*, pages 177–186. ACM.

Hardekopf, B. and Lin, C. (2007). The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *In PLDI*, pages 290–299. ACM.

Hardekopf, B. and Lin, C. (2011). Flow-sensitive pointer analysis for millions of lines of code. In *In CGO*, pages 265–280.

Hind, M. (2001). Pointer analysis: Haven't we solved this problem yet? In *In PASTE*, pages 54–61. ACM.

Jones, N. D. and Muchnick, S. S. (1982). A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *In POPL*, pages 66–74. ACM.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Maleej, M., Paisante, V., Ramos, P., Gonnord, L., and Pereira, F. M. Q. (2017). Pointer disambiguation via strict inequalities. In *CGO*. ACM.

Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., and Pereira, F. M. Q. (2014). Validation of memory accesses through symbolic analyses. In *In OOPSLA*, pages 791–809. ACM.

Paisante, V., Maleej, M., Gonnord, L., Barbosa, L., and Pereira, F. M. Q. (2016). Symbolic range analysis of pointer. In *CGO*. ACM.

Paisante, V. M., Rodrigues, R. E., Saggioro, L. F. Z., e Oliveria, L. B., and Pereira, F. M. Q. (2014). Prevencao de ataques em sistemas distribuidos via analise de intervalos. In *Anais do SBSeg*, page 209.

Pearce, D. J., Kelly, P. H. J., and Hankin, C. (2004). Efficient field-sensitive pointer analysis for C. In *In PASTE*, pages 37–42.

Rugina, R. and Rinard, M. C. (2005). Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27(2):185–235.

Ryder, B. G., Landi, W. A., Stocks, P. A., Zhang, S., and Altucher, R. (2001). A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186.

Saggioro, L. F. Z., Paisante, V. M., Rodrigues, R. E., e Oliveria, L. B., and Pereira, F. M. Q. (2015). Cruzando dados distribudos para detectar estouro de inteiro. volume 13. IEEE.

Sagiv, M., Reps, T., and Wilhelm, R. (1998). Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50.

Shang, L., Xie, X., and Xue, J. (2012). On-demand dynamic symmary-based points-to analysis. In *In CGO*, pages 264–274. ACM.

Wolfe, M. (1996). *High Performance Compilers for Parallel Computing*. Adison-Wesley, 1st edition.

Yan, D., Xu, G., and Rountev, A. (2011). Demand-driven context-sensitive alias analysis for java. In *In ISSTA*, pages 155–165. ACM.

Yong, S. H. and Horwitz, S. (2004). Pointer-range analysis. In *In SAS*, pages 133–148. Springer.

Zhang, Q., Xiao, X., Zhang, C., Yuan, H., and Su, Z. (2014). Efficient subcubic alias analysis for C. In *In OOPSLA*, pages 829–845. ACM.

Zhao, Q., Rabbah, R., and Wong, W.-F. (2005). Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32.